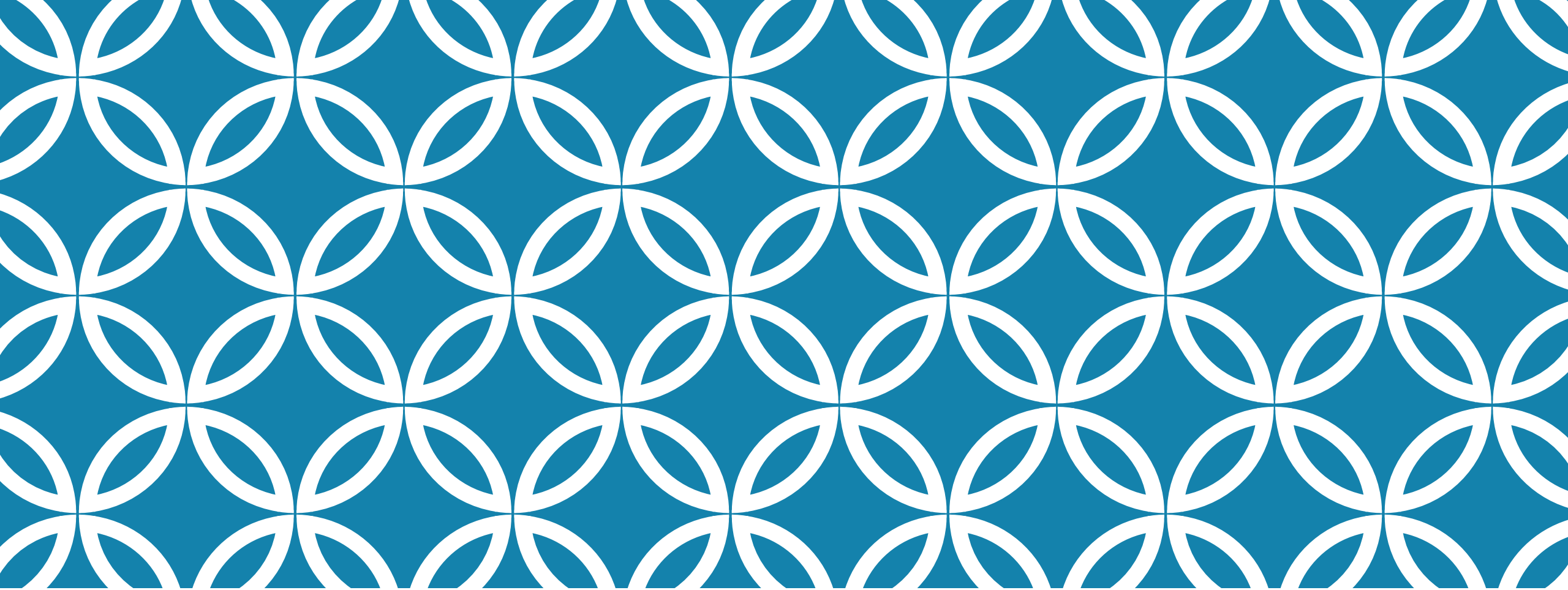


COMPARING FORMALIZATIONS OF PROOFS ABOUT PROGRAMMING LANGUAGES

Yanjun Yang ‘20
Advised by Prof. David Walker and Matthew Weaver

MOTIVATION AND GOAL

Type safety

- Well-typed language do not “go wrong”
- Proofs of type safety can be complex

Goal: Explore ways to formalize these proofs, and compare how different formalizations affect type safety proofs

PROBLEM BACKGROUND AND RELATED WORK

Simply Typed Lambda Calculus ($\lambda^{\rightarrow}$) (Church 1940)

- First typed programming language
- Type-safe → can be used as a model of formal logic

Types and Programming Languages (Pierce 2002)

- Outlines general proofs of type safety for PL including $\lambda^{\rightarrow}$

APPROACH

Explore proofs of type safety using different formulations of $\lambda\!\rightarrow$

Formalize and verify proofs with Agda

(UNTYPED) LAMBDA CALCULUS

Simple language with three kinds of expressions (e):

- Variables: x, y, etc.
- Functions: $\lambda x.e$ (i.e. $f(x) = e$)
- Applications: $e_1\ e_2$ (i.e. $e_1(e_2)$, where e_1 is a function)

SIMPLY TYPED LAMBDA CALCULUS

One additional feature: Bound variables have types (τ)

- Functions: λx.e becomes λx:τ.e

SIMPLY TYPED LAMBDA CALCULUS

One additional feature: Bound variables have types (τ)

- Functions: λx.e becomes λx:τ.e

Functions must be called with arguments of the correct type

- (λx:int.x + 1) 3 →* 4
- (λx:int.x + 1) true →* Error

SIMPLY TYPED LAMBDA CALCULUS

Emergent property: Terms can have types if they are well formed

- (λx:int.x + 1) 3 : int because (λx:int.x + 1) 3 →* 4, and 4 : int

SIMPLY TYPED LAMBDA CALCULUS

Emergent property: Terms can have types if they are well formed

- (λx:int.x + 1) 3 : int because (λx:int.x + 1) 3 →* 4, and 4 : int
- (λx:int.x + 1) : int ⟶ int

SIMPLY TYPED LAMBDA CALCULUS

Type safety: Well-typed terms always evaluate to other well-typed terms of the same type

- $(\lambda x{:}int.x + 1)\ 3 : int \rightarrow^* 4 : int$

SIMPLY TYPED LAMBDA CALCULUS

Type safety: Well-typed terms always evaluate to other well-typed terms of the same type

- (λx:int.x + 1) 3 : int →* 4 : int
- ~~(λx:int.x + 1) 3 : int →* true : bool~~

SIMPLY TYPED LAMBDA CALCULUS

Type safety: Well-typed terms always evaluate to other well-typed terms of the same type

- (λx:int.x + 1) 3 : int →* 4 : int
- ~~(λx:int.x + 1) 3 : int →* true : bool~~
- ~~(λx:int.x + 1) 3 : int →* Error~~

IMPLEMENTATION

Formulations of $\lambda^{\rightarrow}$ in Agda:

- Extrinsically typed $\lambda^{\rightarrow}$ with named variables
- Extrinsically typed $\lambda^{\rightarrow}$ with nameless variables
- Intrinsically typed $\lambda^{\rightarrow}$ with nameless variables

EXTRINSICALLY TYPED, NAMED

Extrinsically typed: Typing of terms is done independently of the definition of the term

Named variables: Variables are referred to by the name they are given when bound

- λx:int.x + 1

EXTRINSICALLY TYPED, NAMED

```
data Type : Set where
    Boolean : Type
    Function : Type → Type → Type
data Term : Set where
    True : Term
    False : Term
    Var : Name → Term
    Fun : Name → Type → Term → Term
    App : Term → Term → Term
```

```
data Type-Proof (Γ : Context) : Term → Type
→ Set where
    Type-True : Type-Proof Γ True Boolean
    Type-False : Type-Proof Γ False Boolean
    Type-Var : (n : Name) (t : Type) (p : (n , t)
∈ Γ) → Type-Proof Γ (Var n) t
    Type-Fun : (n : Name) (t t′ : Type) (e :
Term) → Type-Proof ((n , t) :: Γ) e t′ → Type-
Proof Γ (Fun n t e) (Function t t′)
    Type-App : (t t′ : Type) (e₁ e₂ : Term) →
Type-Proof Γ e₁ (Function t t′) → Type-Proof
Γ e₂ t → Type-Proof Γ (App e₁ e₂) t′
```

EXTRINSICALLY TYPED, NAMELESS

Nameless Variables: Variables are referred to by some intrinsic property

- De Brujin index: Number of binding sites away
- λx:int.x + 1 → λ_:int.[1] + 1

EXTRINSICALLY TYPED, NAMELESS

Nameless Variables: Variables are referred to by some intrinsic property

- De Brujin index: Number of binding sites away
- λx:int.x + 1 → λ_:int.[1] + 1
- λx:int.λy:int.x + y → λ_:int. λ_:int.[2] + [1]

EXTRINSICALLY TYPED, NAMELESS

Nameless Variables: Variables are referred to by some intrinsic property

- De Brujin index: Number of binding sites away
- λx:int.x + 1 → λ_:int.[1] + 1
- λx:int.λy:int.x + y → λ_:int. λ_:int.[2] + [1]
- λx:int.x + ((λy:int.x + y) x) → λ_:int.[1] + ((λ_:int.[2] + [1]) [1])

EXTRINSICALLY TYPED, NAMELESS

```
data Type : Set where
    Boolean : Type
    Function : Type → Type → Type
data Term (Γ : Context) : Set where
  Var : Variable Γ → Term Γ
  Fun : (t : Type) → Term (Γ , t) → Term Γ
  App : Term Γ → Term Γ → Term Γ
  True : Term Γ
  False : Term Γ
```

```
data Type-Proof (Γ : Context) : Term Γ →
Type → Set where
    Type-True : Type-Proof Γ True Boolean
    Type-False : Type-Proof Γ False Boolean
    Type-Var : ∀ v → Type-Proof Γ (Var v)
(type-var Γ v)
    Type-Fun : (t : Type) (e : Term (Γ , t)) (t′ :
Type) → Type-Proof (Γ , t) e t′ → Type-Proof
Γ (Fun t e) (Function t t′)
    Type-App : (t₁ : Type) (e₁ : Term Γ) (t₂ :
Type) (e₂ : Term Γ) → Type-Proof Γ e₁
(Function t₁ t₂) → Type-Proof Γ e₂ t₁ →
Type-Proof Γ (App e₁ e₂) t₂
```

INTRINSICALLY TYPED, NAMELESS

Intrinsically typed: Terms inherently contain their proof of well-typedness

- (λx:int.x + 1) 3 → (((λ_:int.([1] : int) + (1 : int)) : int ⟶ int) (3 : int)) : int

INTRINSICALLY TYPED, NAMELESS

Intrinsically typed: Terms inherently contain their proof of well-typedness

- (λx:int.x + 1) 3 → (((λ_:int.([1] : int) + (1 : int)) : int → int) (3 : int)) : int
- (λx:int.x + 1) true → Impossible

INTRINSICALLY TYPED, NAMELESS

Intrinsically typed: Terms inherently contain their proof of well-typedness

- (λx:int.x + 1) 3 → (((λ_:int.([1] : int) + (1 : int)) : int ⟶ int) (3 : int)) : int
- (λx:int.x + 1) true → Impossible

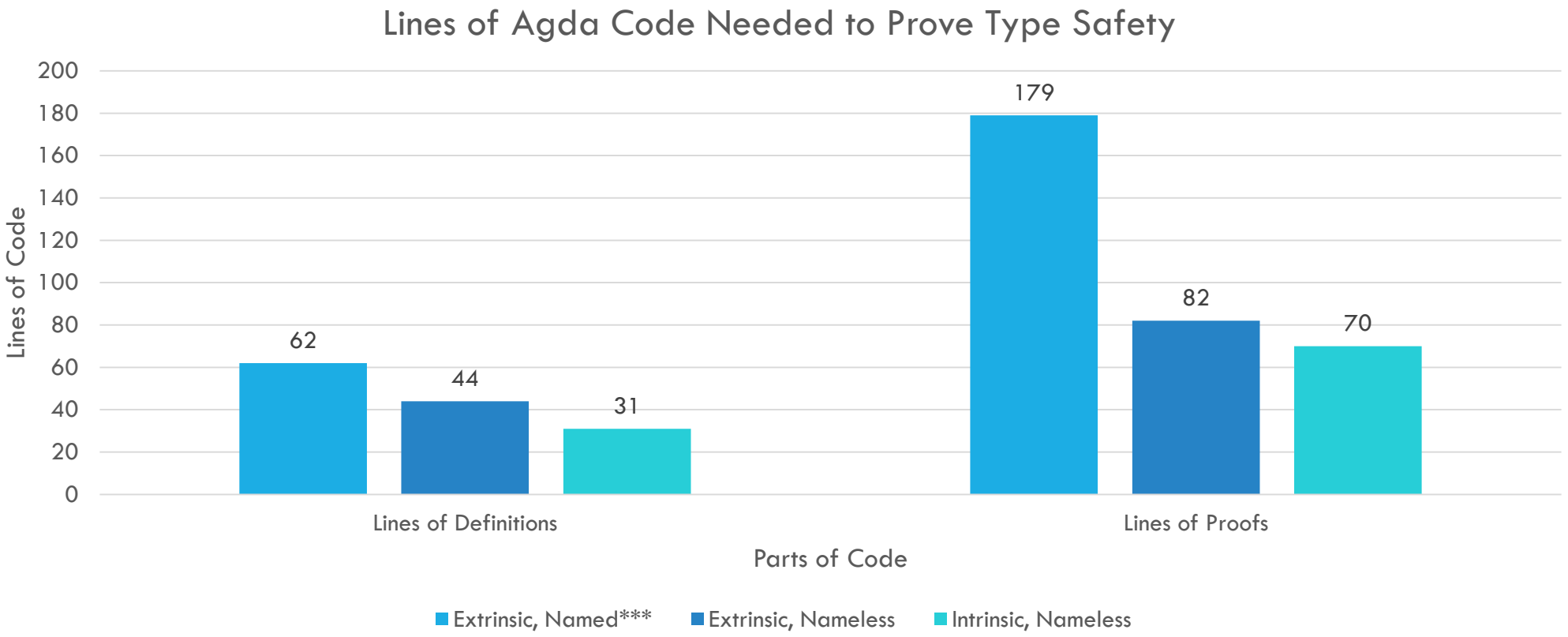
Emergent Property: All terms are well-typed.

INTRINSICALLY TYPED, NAMELESS

```
data Type : Set where
    Boolean : Type
    Function : Type → Type → Type
data Term (Γ : Context) : Type → Set where
    Var : (t : Type) (v : Variable Γ) → (type-var
Γ v ≡ t) → Term Γ t
    Fun : (t t′ : Type) → Term (Γ , t) t′ → Term Γ
(Function t t′)
    App : (t t′ : Type) → Term Γ (Function t t′) →
Term Γ t → Term Γ t′
    True : Term Γ Boolean
    False : Term Γ Boolean
```

No Type-Proof needed

RESULTS



*** Proof only applies to call-by-value evaluation order

DISCUSSION: PROS AND CONS

Extrinsically Typed

Terms are lightweight

Intrinsically Typed

Terms are syntactically bulkier

DISCUSSION: PROS AND CONS

Extrinsically Typed

Terms are lightweight

External typing judgement needed, leading to more difficult proofs

Intrinsically Typed

Terms are syntactically bulkier

No external typing judgment necessary, simpler proofs

DISCUSSION: PROS AND CONS

Named Variables

Variable shadowing

Nameless Variables

No variable shadowing

DISCUSSION: PROS AND CONS

Named Variables

Variable shadowing

Variable capture

Nameless Variables

No variable shadowing

No variable capture

DISCUSSION: PROS AND CONS

Named Variables

Variable shadowing

Variable capture

Easy to program in

Nameless Variables

No variable shadowing

No variable capture

Difficult/awkward to program in

CONCLUSION

Compared 3 different formulations of $\lambda^{\rightarrow}$ and their respective type-safety proofs in Agda

Found that intrinsic typing and nameless variable are easier in proofs

Found that extrinsic typing and named variable are easier in programming

Future work: Use Cubical Type Theory to make named variables easier to use in proofs

THE END

Thank you!

ACKNOWLEDGEMENTS

I would like to thank Matthew Weaver and Prof. David Walker for advising me on this project throughout this semester.